

# COMP 322: Fundamentals of Parallel Programming

## Lecture 37: Algorithms based on Parallel Prefix (Scan) operations, cont.

Mack Joyner and Zoran Budimlić  
{mjoyner, zoran}@rice.edu

### Acknowledgements:

- Book chapter on “Prefix Sums and Their Applications”, Guy E. Blelloch, CMU
- Slides on “Parallel prefix adders”, Kostas Vitoroulis, Concordia University

<http://comp322.rice.edu>



# Worksheet #36 problem statement: Parallelizing the Split step in Radix Sort

The Radix Sort algorithm loops over the bits in the binary representation of the keys, starting at the lowest bit, and executes a split operation for each bit as shown below. The split operation packs the keys with a 0 in the corresponding bit to the bottom of a vector, and packs the keys with a 1 to the top of the same vector. It maintains the order within both groups. The sort works because each split operation sorts the keys with respect to the current bit and maintains the sorted order of all the lower bits. Your task is to show how the split operation can be performed in parallel using scan, reverse, not operations, and to explain your answer.

```

                                [101 111 011 001 100 010 111 010]
1. A =                          [5 7 3 1 4 2 7 2]
2. A<0> =                        [1 1 1 1 0 0 1 0] //lowest bit
3. A ← split(A, A<0>) =         [4 2 2 5 7 3 1 7]
4. A<1> =                        [0 1 1 0 1 1 0 1] // middle bit
5. A ← split(A, A<1>) =         [4 5 1 2 2 7 3 7]
6. A<2> =                        [1 1 0 0 0 1 0 1] // highest bit
7. A ← split(A, A<2>) =         [1 2 2 3 4 5 7 7]
```



# Worksheet #36 solution: Parallelizing the Split step in Radix Sort

|  |   |   |   |   |   |   |   |   |   |    |
|--|---|---|---|---|---|---|---|---|---|----|
| <pre> procedure split(A, Flags)   I-down ← prescan(+, not(Flags)) // prescan = exclusive prefix sum   I-up   ← rev(n - scan(+, rev(Flags))) // rev = reverse   in parallel for each index i     if (Flags[i])       Index[i] ← I-up[i]     else       Index[i] ← I-down[i]   result ← permute(A, Index) </pre> |   |   |   |   |   |   |   |   |   |    |
| A  | = | [ | 5 | 7 | 3 | 1 | 4 | 2 | 7 | 2] |
| Flags  | = | [ | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0] |
| I-down   | = | [ | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 2] |
| I-up   | = | [ | 3 | 4 | 5 | 6 | 7 | 7 | 7 | 8] |
| Index  | = | [ | 3 | 4 | 5 | 6 | 0 | 1 | 7 | 2] |
| permute(A, Index)  | = | [ | 4 | 2 | 2 | 5 | 7 | 3 | 1 | 7] |

FIGURE 1.9

The `split` operation packs the elements with a 0 in the corresponding flag position to the bottom of a vector, and packs the elements with a 1 to the top of the same vector. The `permute` writes each element of `A` to the index specified by the corresponding position in `Index`.



# Parallelizing Prefix Sum (Lecture 13)

---

Observation: each prefix sum can be decomposed into reusable terms of power-of-2-size e.g.

$$\begin{aligned} X[6] &= A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6] \\ &= (A[0] + A[1] + A[2] + A[3]) + (A[4] + A[5]) + A[6] \end{aligned}$$

Approach:

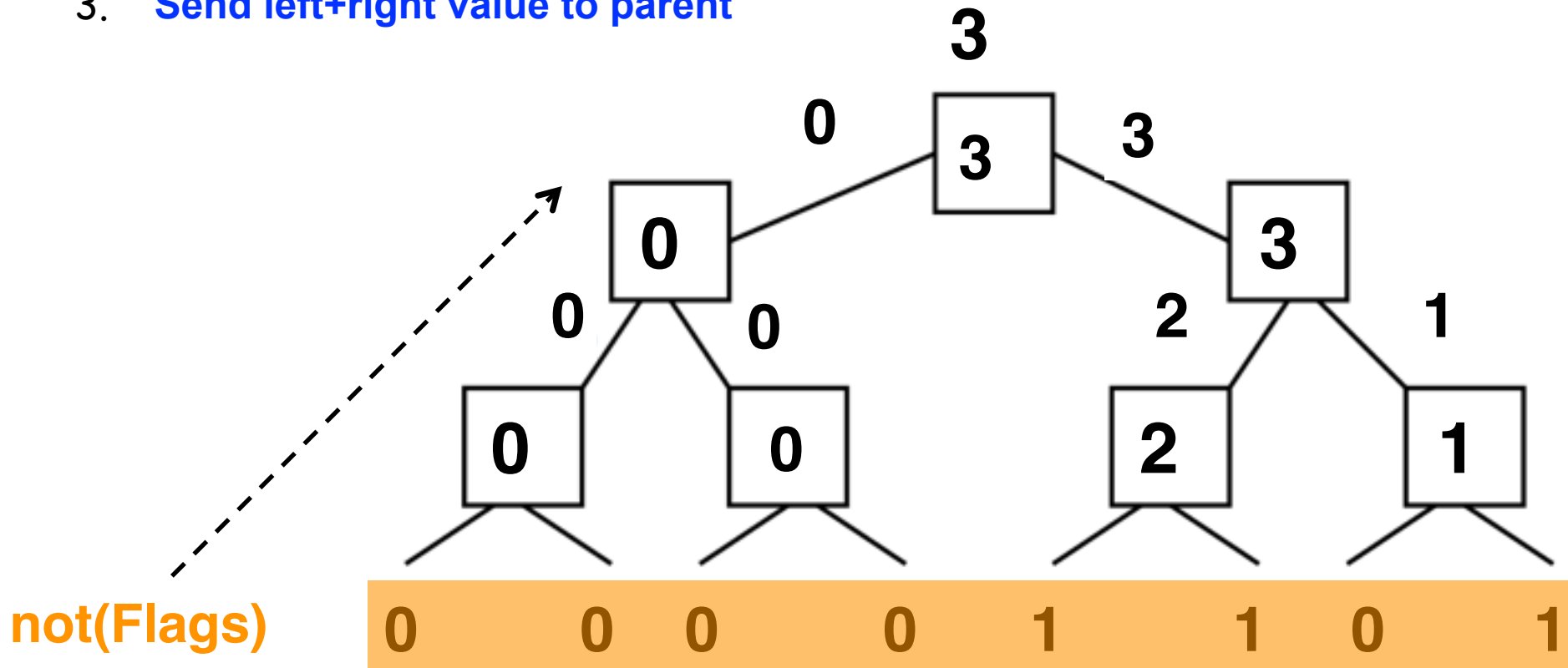
- Combine reduction tree idea from Parallel Array Sum with partial sum idea from Sequential Prefix Sum
- Use an “upward sweep” to perform parallel reduction, while storing partial sum terms in tree nodes
- Use a “downward sweep” to compute prefix sums while reusing partial sum terms stored in upward sweep



# Parallel Pre-scan Sum: Upward Sweep

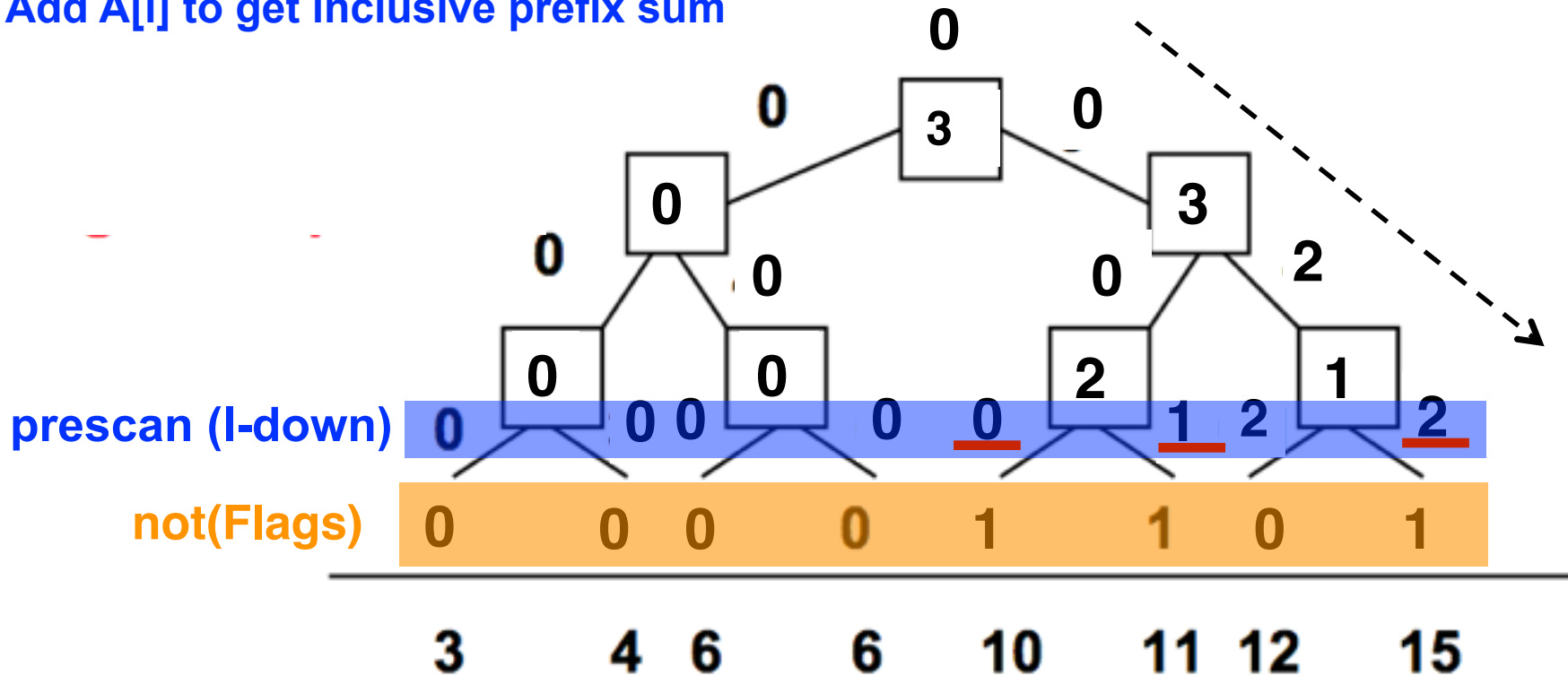
Upward sweep is just like Parallel Reduction, except that partial sums are also stored along the way

1. Receive values from left and right children
2. Compute left+right and store in box
3. Send left+right value to parent

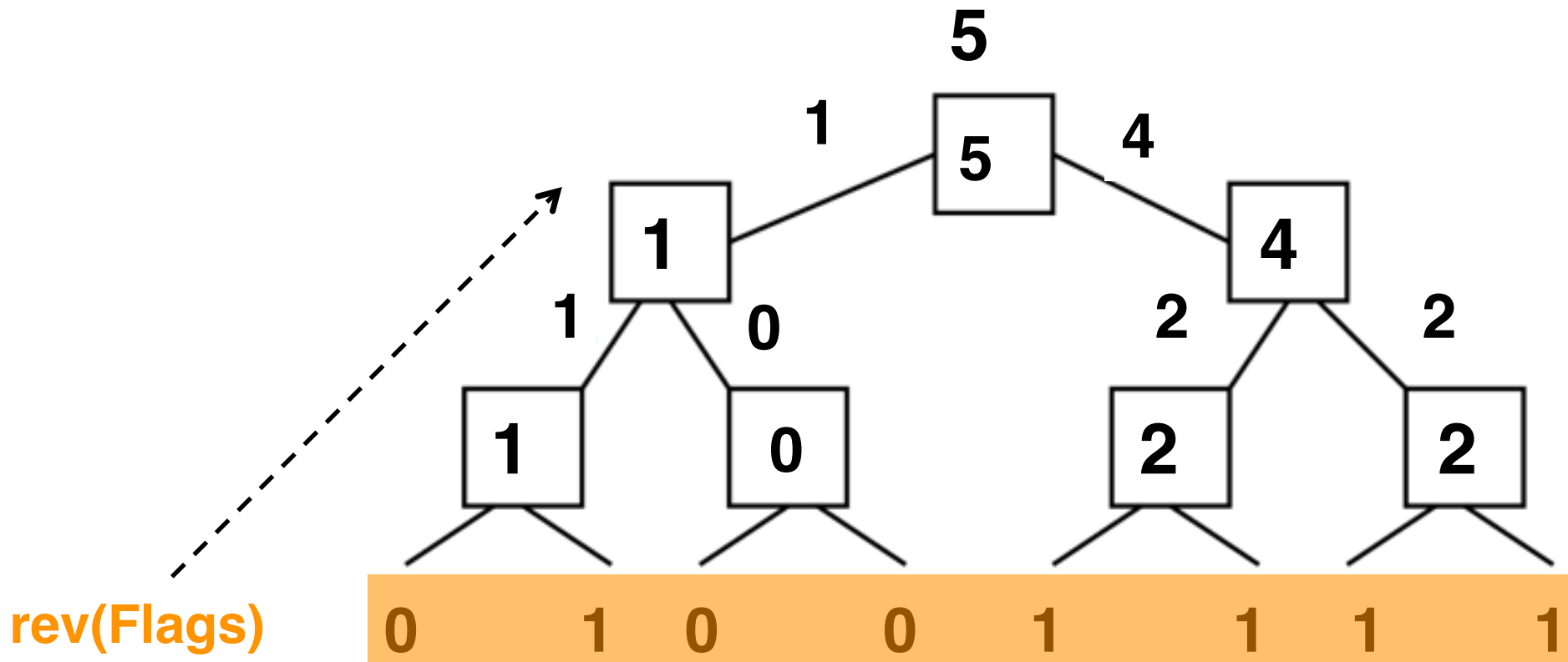


# Parallel Pre-scan Sum: Downward Sweep

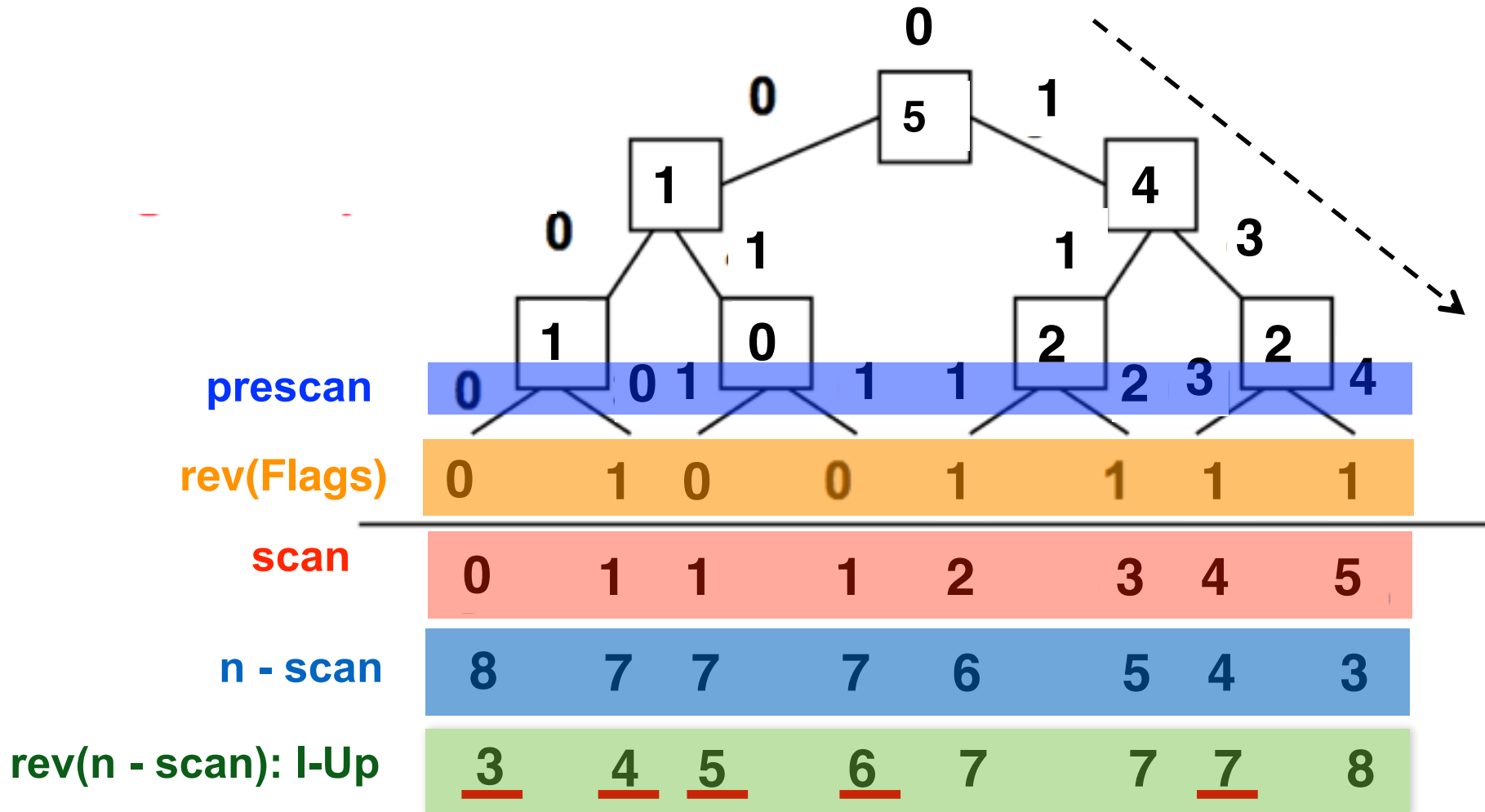
1. Receive value from parent (root receives 0)
2. Send parent's value to LEFT child (prefix sum for elements to left of left child's subtree)
3. Send parent's value+ left child's box value to RIGHT child (prefix sum for elements to left of right child's subtree)
4. Add  $A[i]$  to get inclusive prefix sum



# Parallel Scan Sum: Upward Sweep



# Parallel Scan Sum: Downward Sweep





# Worksheet #36 solution: Parallelizing the Split step in Radix Sort

```

procedure split(A, Flags)
  I-down ← prescan(+, not(Flags)) // prescan = exclusive prefix sum
  I-up   ← rev(n - scan(+, rev(Flags))) // rev = reverse
  in parallel for each index i
    if (Flags[i])
      Index[i] ← I-up[i]
    else
      Index[i] ← I-down[i]
  result ← permute(A, Index)

```

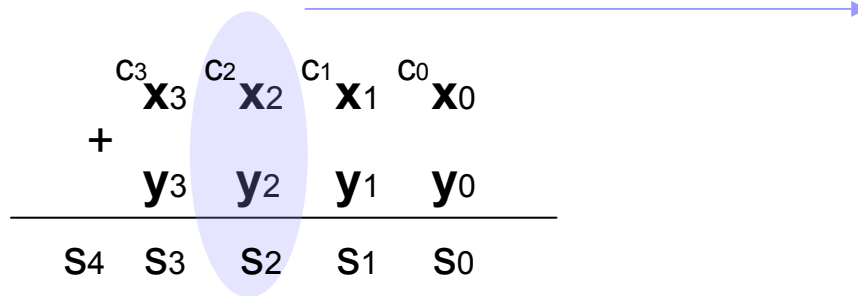
|                   |   |  |
|-------------------|---|--|
| A                 | = | [ 5    7    3    1    4    2    7    2 ] |
| Flags             | = | [ 1    1    1    1    0    0    1    0 ] |
| I-down            | = | [ 0    0    0    0    0    1    2    2 ] |
| I-up              | = | [ 3    4    5    6    7    7    7    8 ] |
| Index             | = | [ 3    4    5    6    0    1    7    2 ] |
| permute(A, Index) | = | [ 4    2    2    5    7    3    1    7 ] |

FIGURE 1.9

The `split` operation packs the elements with a 0 in the corresponding flag position to the bottom of a vector, and packs the elements with a 1 to the top of the same vector. The `permute` writes each element of `A` to the index specified by the corresponding position in `Index`.



# Binary Addition



This is the pen and paper addition of two 4-bit binary numbers  $\mathbf{x}$  and  $\mathbf{y}$ .  
 $\mathbf{c}$  represents the generated carries.  
 $\mathbf{s}$  represents the produced sum bits.

A **stage** of the addition is the set of  $\mathbf{x}$  and  $\mathbf{y}$  bits being used to produce the appropriate sum and carry bits. For example the highlighted bits  $\mathbf{x}_2$ ,  $\mathbf{y}_2$  constitute **stage 2** which generates carry  $\mathbf{c}_2$  and sum  $\mathbf{s}_2$ .

Each stage  $i$  adds bits  $a_i$ ,  $b_i$ ,  $c_{i-1}$  and produces bits  $s_i$ ,  $c_i$   
 The following hold:

| $a_i$ | $b_i$ | $c_i$     | Comment:                                 | Formal definition:                       |
|-------|-------|-----------|--|--|
| 0     | 0     | 0         | The stage "kills" an incoming carry.     | "Kill" bit: $k_i = \overline{x_i + y_i}$ |
| 0     | 1     | $c_{i-1}$ | The stage "propagates" an incoming carry | "Propagate" bit: $p_i = x_i \oplus y_i$  |
| 1     | 0     | $c_{i-1}$ | The stage "propagates" an incoming carry |  |
| 1     | 1     | 1         | The stage "generates" a carry out        | "Generate" bit: $g_i = x_i \bullet y_i$  |

# Binary Addition

| $a_i$ | $b_i$ | $c_i$     | Comment:                                 | Formal definition:                         |
|-------|-------|-----------|--|--|
| 0     | 0     | 0         | The stage “kills” an incoming carry.     | “Kill” bit: $k_i = \overline{x_i + y_i}$   |
| 0     | 1     | $c_{i-1}$ | The stage “propagates” an incoming carry | “Propagate” bit:<br>$p_i = x_i \oplus y_i$ |
| 1     | 0     | $c_{i-1}$ | The stage “propagates” an incoming carry |  |
| 1     | 1     | 1         | The stage “generates” a carry out        | “Generate” bit: $g_i = x_i \bullet y_i$    |

The carry  $c_i$  generated by a stage  $i$  is given by the equation:

$$c_i = g_i + p_i \cdot c_{i-1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_{i-1}$$

This equation can be simplified to:

$$c_i = x_i \cdot y_i + (x_i + y_i) \cdot c_{i-1} = g_i + a_i \cdot c_{i-1}$$

The “a” term in the equation being the “alive” bit.

The later form of the equation uses an OR gate instead of an XOR which is a more efficient gate when implemented in CMOS technology. Note that:

$$a_i = \overline{k_i}$$

Where  $k_i$  is the “kill” bit defined in the table above.

# Binary addition as a prefix sum problem.

- We define a new operator: “  $\circ$  ”
- Input is a vector of pairs of ‘propagate’ and ‘generate’ bits:

$$(g_n, p_n)(g_{n-1}, p_{n-1}) \dots (g_0, p_0)$$

- Output is a new vector of pairs:

$$(G_n, P_n)(G_{n-1}, P_{n-1}) \dots (G_0, P_0)$$

- Each pair of the output vector is calculated by the following definition:

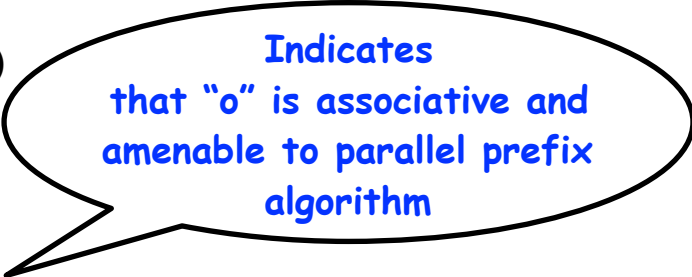
$$(G_i, P_i) = (g_i, p_i) \circ (G_{i-1}, P_{i-1})$$

Where:

$$(G_0, P_0) = (g_0, p_0)$$

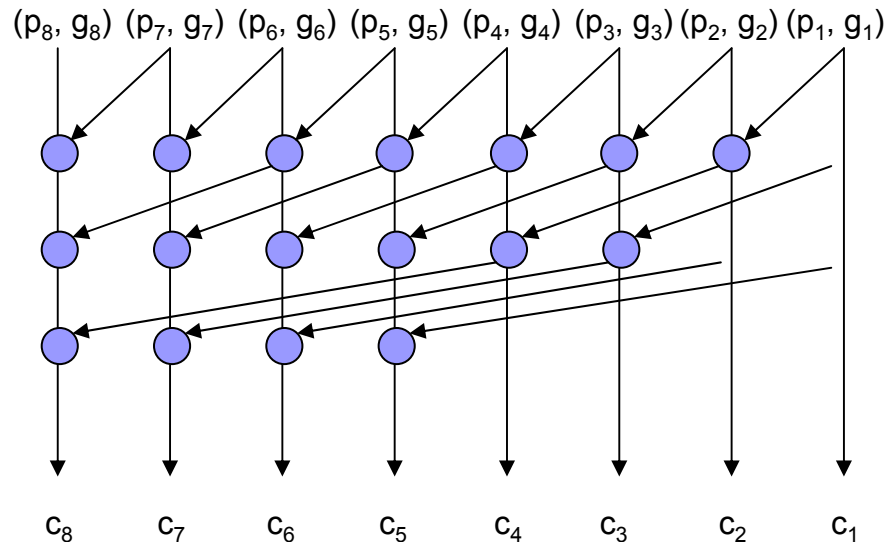
$$(g_x, p_x) \circ (g_y, p_y) = (g_x + p_x \cdot g_y, p_x \cdot p_y)$$

with  $+$ ,  $\cdot$  being the OR, AND operations



Indicates that “o” is associative and amenable to parallel prefix algorithm

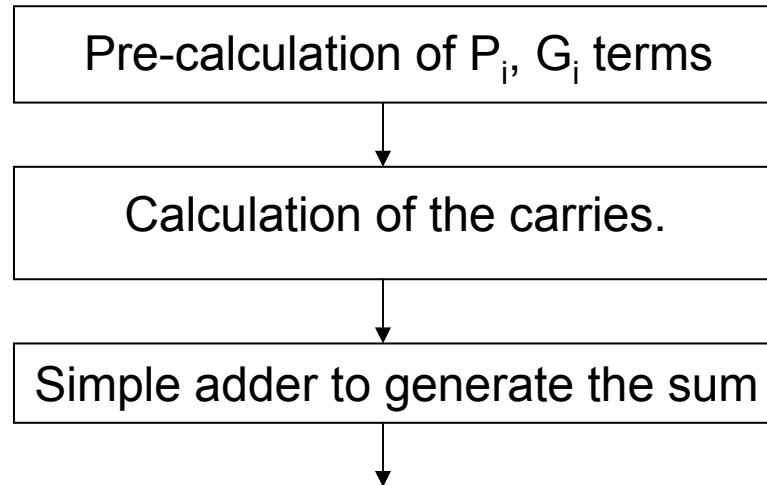
# 1973: Kogge-Stone adder



- The Kogge-Stone adder has:
  - Low depth
  - High node count (implies more area).
  - Minimal fan-out of 1 at each node (implies faster performance).

# Summary

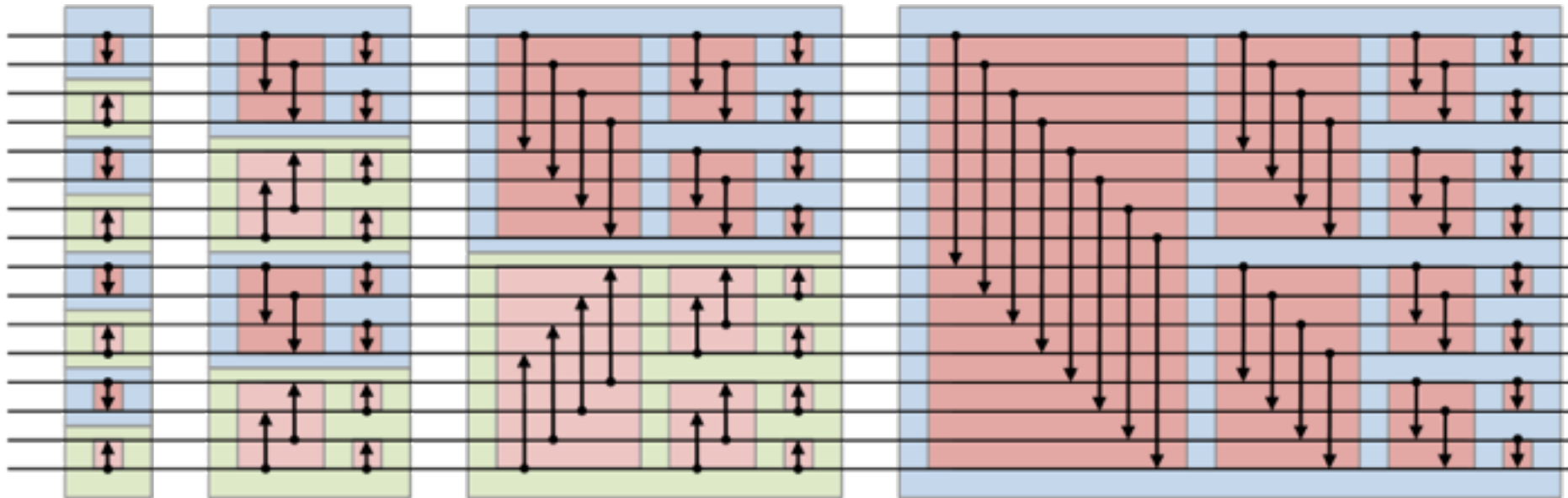
- A parallel prefix adder can be seen as a 3-stage process:



- There exist various architectures for the carry calculation part.
- Trade-offs in these architectures involve the
  - area of the adder
  - its depth
  - the fan-out of the nodes
  - the overall wiring network.

# Parallel Algorithms, Computation Graphs, and Circuits

- Today's lecture shows that parallel algorithms, computation graphs, and circuits represent different approaches to *parallel computational thinking*
- A parallel algorithm unfolds into a computation graph when executing
- A circuit represents an “unrolled” computation graph in hardware e.g., see bitonic sorting network in [https://en.wikipedia.org/wiki/Bitonic\\_sorter](https://en.wikipedia.org/wiki/Bitonic_sorter)



# Announcements & Reminders

---

- **HW5 is due Friday, April 19th with automatic extension until Sunday, April 21st**
  - **You may use any remaining slip days after the extension**
- **Optional quiz available for Unit 10**
  - **Also due April 19th**
  - **We will pick the best 9 of your 10 quiz scores**
- **No lab on Thursday**
- **April 19th: Course review (scope of Exam 2) in class**





# Worksheet #37: Creating a Circuit for Parallel Prefix Sums

Name: \_\_\_\_\_

Netid: \_\_\_\_\_

Assume that you have a full adder cell,  $\oplus$ , that can be used as a building block for circuits (no need to worry about carry's). Create a circuit that generates the prefix sums for 1, ... 6, by adding at most 5 more cells to the sketch shown below, while ensuring that the CPL is at most 3 cells long. Assume that you can duplicate any value (fan-out) to whatever degree you like without any penalty.

